State Management

Play+ State Management Helper Introduction ■ In the Play+ ecosystem, application state should be predictable, resilient, and easy to reason about. This guide is based on the concept of Resilience by Abstraction , providing a standardized way to handle local, global, and asynchronous state. A well-architected state management system is crucial for building complex, data-driven applications. Our approach establishes consistent patterns that align with our core design pillars: creating a Distinct and readable architecture, providing an Intuitive API that minimizes boilerplate, and supporting an Inclusive experience by enabling accessible and observable state changes. Package Info ■ The Play+ state management helpers and patterns are integrated into the Golden Path starter kit. For existing projects, the core utilities can be installed via a dedicated package. Package / Path Description Golden Path (Recommended) Pre-installed (/src/stores or /src/app/core) Uplift Path npm install @playplus/state Folder Reference ■ State management follows our standardized folder structure, separating global state logic from component or feature logic. File / Directory Purpose & Guidelines src/stores/ (React) The recommended location for global Zustand store definitions (e.g., auth.store.ts). src/app/core/services/ (Angular) The recommended location for stateful services that use RxJS Subjects. config/play.state.config.json An optional file for overriding default state management behaviors and linting rules. Helper - Pillars Alignment ■ A predictable state management strategy is fundamental to our design philosophy. Pillar How This Helper Aligns Intuitive Primary Pillar: Abstracts the complexity of libraries like Zustand and RxJS into a simple, predictable pattern. Distinct Enforces a consistent, structured state architecture across all applications, making them easier to navigate and maintain. Adaptive A well-managed state allows the UI to react fluidly and reliably to data changes, adapting to user interactions seamlessly. Helper Overview ■ The Play+ state management solution provides a set of patterns and a smart store factory (createPlayStore) to abstract the plumbing of state management. Instead of setting up stores from scratch, developers use our pre-configured helper that bakes in best practices for immutability, performance, and debuggability. It automates and simplifies: Store Creation : A single function, createPlayStore , sets up a global store with middleware for logging and immutability checks. Immutability : Automatically uses immer behind the scenes to prevent direct state mutations, a common source of bugs. CI/CD Validation : The toolchain includes scripts to lint for common state management pitfalls, such as storing derived state or creating un-optimized selectors. Automated Logging : When integrated with playlog , all state mutations can be automatically logged, providing a clear audit trail for debugging. The goal is for developers to define their state shape and actions, and trust that the system is handling the underlying complexity and enforcement correctly. Config Options ■ Optional overrides for state management behavior can be placed in config/play.state.config.json . Config Variable Default Value Description Recommended Value storeNaming PascalCase Enforces a naming convention for store files (e.g., AuthStore.ts). PascalCase enforceSelectors true If true, the linter will warn against selecting the entire state

object in components. true allowDerivedState false If false, the linter will flag instances where derived data is stored in state. false immutability strict The level of immutability enforcement. strict uses immer . strict Helper Methods ■ Core Methods ■ Method Name Description Signature createPlayStore Factory function to create Zustand stores with Play+ middleware. createPlayStore<T>(initializer: StateCreator<T>, options?: { debug: boolean }): StoreApi<T> createStateService Factory function to create Angular state services with RxJS. createStateService<T>(initialState: T): StateService<T> withPlayMiddleware Adds Play+ middleware to existing stores. withPlayMiddleware<T>(store: StoreApi<T>): StoreApi<T> validateState Validates state structure and immutability. validateState<T>(state: T, schema: StateSchema): ValidationResult Angular Integration ■ PlayStateService ■ Angular service wrapper that integrates with Play+ logging and provides component-specific state management utilities. import { PlayStateService } from '../services/playstate.service' ; @ Component ({ ... }) export class MyComponent { constructor (private playState : PlayStateService) { } ngOnInit ( ) { // Subscribe to state changes this . playState . select ('user') . subscribe (user => { this . currentUser = user ; }) ; } updateUser (user : User) { // Dispatch state change this . playState . dispatch ('setUser' , user) ; } } State Directive ■ Automatically manages component state and provides utilities. <!-- Manage component state --> < div playState = " user-profile " [stateConfig] = " userStateConfig " > < app-user-form > </ app-user-form > </ div > State Pipe ■ Provides state utilities in templates. <!-- Select state in template --> < div > </ div > <!-- Transform state --> < div > </ div > React: Creating and Using a Global Auth Store ■ // src/stores/auth.store.ts import { createPlayStore } from "@playplus/state" ; import { User } from "../types" ; interface AuthState { user : User | null ; isAuthenticated : boolean ; setUser : (user : User | null) => void ; } // Define the store using the helper export const useAuthStore = createPlayStore < AuthState > ((set) => ({ user : null , isAuthenticated : false , setUser : (user) => set ({ user : user , isAuthenticated : ! ! user , }) , })) ; // src/components/LoginButton.tsx import { useAuthStore } from "../stores/auth.store" ; function LoginButton ( ) { // Use a selector to get only the 'setUser' action to prevent unnecessary re-renders const setUser = useAuthStore ((state) => state . setUser) ; const handleLogin = ( ) => { const fakeUser = { id : "1" , name : "Jane Doe" } ; setUser (fakeUser) ; } ; return < button onClick = { handleLogin } > Log In </ button > ; } Angular: A Stateful Service with RxJS ■ // src/app/core/services/auth.store.service.ts import { Injectable } from "@angular/core" ; import { BehaviorSubject , map } from "rxjs" ; import { User } from "../models" ; interface AuthState { user : User | null ; isAuthenticated : boolean ; } @ Injectable ({ providedIn : "root" }) export class AuthStoreService { private readonly state$ = new BehaviorSubject < AuthState > ({ user : null , isAuthenticated : false , }) ; // Expose state as observables readonly user$ = this . state$. pipe (map ((s) => s . user)) ; readonly isAuthenticated$ = this . state$. pipe (map ((s) => s . isAuthenticated)) ; // Actions to mutate state setUser (user : User | null) : void { this . state$. next ({ user , isAuthenticated : ! ! user , }) ; } } import { createPlayStore } from "@playplus/state" ; // Create a simple counter store interface CounterState { count : number ; increment : ( ) => void ; decrement : ( ) => void ; reset : ( ) => void ; } export const useCounterStore = createPlayStore < CounterState > ((set) => ({ count : 0 , increment : ( ) =>

set ((state) => ({ count : state . count + 1 })) , decrement : ( ) => set ((state) => ({ count : state . count - 1 })) , reset : ( ) => set ({ count : 0 }) , })) ; // Use in component function Counter ( ) { const { count , increment , decrement , reset } = useCounterStore ( ) ; return (< div > < p > Count : { count } < / p > < button onClick = { increment } > + < / button > < button onClick = { decrement } > - < / button > < button onClick = { reset } > Reset < / button > < / div >) ; } Additional Info ■ Why We Created This Helper ■ State management is one of the most complex parts of modern web development. Without a standardized approach, projects can suffer from: Inconsistent patterns across different features. Bugs from direct state mutation. Poor performance from un-optimized component re-renders. Difficulty debugging state changes. The Play+ state management helper provides an opinionated, production-ready pattern that solves these problems. It abstracts the boilerplate of setting up robust stores and provides automated checks, allowing developers to manage state confidently and consistently. State Management Principles ■ 1. Single Source of Truth ■ Each piece of state has a single, authoritative location No duplicate state across different stores or services Clear data flow from source to consumers 2. Immutability ■ State is never mutated directly All changes go through defined actions/methods Automatic immutability enforcement with immer 3. Predictable Updates ■ State changes follow a clear, predictable pattern Actions are the only way to modify state Changes are logged and traceable 4. Performance Optimization ■ Components subscribe only to the state they need Automatic memoization of selectors Efficient re-rendering with shallow equality checks Best Practices ■ DO ■ Use selectors : Subscribe to the smallest piece of state necessary Define actions : All state mutations should go through defined actions Keep state normalized : Avoid nested objects and arrays when possible Use TypeScript : Define interfaces for all state shapes Test state logic : Unit test your state actions and selectors Log state changes : Use playlog to track state mutations DON'T ■ Mutate state directly : Never modify state outside of actions Store derived data : Calculate derived values in components or selectors Create multiple stores for the same domain : Use a single store per domain Subscribe to entire state : Use selectors to get only what you need Ignore performance : Monitor re-renders and optimize selectors Security Considerations ■ State validation : Validate state structure and types Access control : Ensure sensitive state is properly protected Audit logging : Log all state mutations for debugging Error boundaries : Handle state errors gracefully Forbidden Patterns ■ Direct State Mutation ■ // DON'T: Mutate state directly const state = useAuthStore . getState ( ) ; state . user = newUser ; // This will cause issues // DO: Use actions const setUser = useAuthStore . getState ( ) . setUser ; setUser (newUser) ; Storing Derived State ■ // DON'T: Store derived data interface UserState { firstName : string ; lastName : string ; fullName : string ; // Derived from firstName + lastName } // DO: Calculate in component or selector const fullName = useMemo (( ) => ` ${ firstName } ${ lastName } ` , [firstName , lastName]) ; Multiple Stores for Same Domain ■ // DON'T: Create multiple stores for auth const useAuthStore = createPlayStore (...) ; const useUserStore = createPlayStore (...) ; // Duplicate! // DO: Use a single store const useAuthStore = createPlayStore < AuthState > ({ user : null , isAuthenticated : false , setUser : (user) => set ({ user , isAuthenticated : ! ! user }) , }) ; Subscribing to Entire State ■ // DON'T: Subscribe to entire state const state = useAuthStore ( ) ; //

Causes unnecessary re-renders // DO: Use selectors const user = useAuthStore ((state) => state . user) ; const isAuthenticated = useAuthStore ((state) => state . isAuthenticated) ; Required Patterns ■ Use Actions for State Changes ■ // Always use actions to modify state interface CounterState { count : number ; increment : ( ) => void ; decrement : ( ) => void ; } export const useCounterStore = createPlayStore < CounterState > ((set) => ({ count : 0 , increment : ( ) => set ((state) => ({ count : state . count + 1 })) , decrement : ( ) => set ((state) => ({ count : state . count - 1 })) , })) ; Use Selectors for State Access ■ // Always use selectors to access state function UserProfile ( ) { const user = useAuthStore ((state) => state . user) ; const isAuthenticated = useAuthStore ((state) => state . isAuthenticated) ; if (! isAuthenticated) return < LoginPrompt / > ; return < div > Welcome , { user ?. name } ! < / div > ; } Normalize State Structure ■ // Keep state normalized interface AppState { users : Record < string , User > ; posts : Record < string , Post > ; currentUserId : string | null ; } // Instead of nested objects interface BadState { users : User [ ] ; posts : Post [ ] ; } Type Your State ■ // Always define TypeScript interfaces interface AuthState { user : User | null ; isAuthenticated : boolean ; isLoading : boolean ; error : string | null ; setUser : (user : User | null) => void ; setLoading : (loading : boolean) => void ; setError : (error : string | null) => void ; } Testing ■ Unit Testing Stores ■ describe ("AuthStore" , ( ) => { it ("should set user and update authentication status" , ( ) => { const user = { id : "1" , name : "John Doe" } ; act (( ) => { useAuthStore . getState ( ) . setUser (user) ; }) ; const state = useAuthStore . getState ( ) ; expect (state . user) . toEqual (user) ; expect (state . isAuthenticated) . toBe (true) ; }) ; it ("should clear user on logout" , ( ) => { act (( ) => { useAuthStore . getState ( ) . setUser (null) ; }) ; const state = useAuthStore . getState ( ) ; expect (state . user) . toBeNull ( ) ; expect (state . isAuthenticated) . toBe (false) ; }) ; }) ; Integration Testing ■ describe ("UserProfile Component" , ( ) => { it ("should display user information" , ( ) => { const user = { id : "1" , name : "John Doe" } ; useAuthStore . setState ({ user , isAuthenticated : true }) ; render (< UserProfile / >) ; expect (screen . getByText ("John Doe")) . toBeInTheDocument ( ) ; }) ; }) ; Testing Checklist ■ Test all state actions Test state selectors Test error handling Test performance with large state Test integration with components Test state persistence (if applicable) Monitoring and Analytics ■ State Metrics ■ State Size : Monitor the size of state objects Mutation Frequency : Track how often state changes Selector Performance : Monitor selector execution time Re-render Frequency : Track component re-renders Performance Monitoring ■ // Monitor state mutations const originalSetState = useAuthStore . setState ; useAuthStore . setState = (partial , replace) => { const start = performance . now ( ) ; originalSetState (partial , replace) ; const duration = performance . now ( ) - start ; if (duration > 10) { playlog . warn ("Slow state mutation" , { duration , stateKeys : Object . keys (partial) , }) ; } } ; Standards and Enforcement ■ State Integrity ■ Rule Area Description Implementation Details Derived State Never store derived values. Enforced by play:state:check lint script. Singleton Stores Avoid multiple stores for the same domain. Warns on duplicate store IDs during bootstrap. Subscription Boundaries Detect components that re-render too often. Profiler plugin or RxJS scheduler tracing. Security & Stability ■ Area Description Rule IDs / Notes Immutable State Prevent direct mutation of state objects. Enforced by eslint-plugin-immer and immer usage in

createPlayStore . Retry Budget Detect repeated failed state transitions. Async state patterns log failures with counter buckets. Framework-Specific Enforcement ■ React ■ Concern Enforcement Details Rule ID(s) useStore Selector Prevent stale selector traps and excessive re-renders. useShallow or other equality functions are recommended. Suspense Boundaries Required for async-heavy global state. Enforced via a Higher-Order Component (HOC) wrapper. Angular ■ Concern Rule ID(s) / Notes Component Inputs Must use Observables for shared state. strictChangeDetection rule in tsconfig.json . Subject Abuse Flag manual subscriptions that are not unsubscribed. ESLint plugin-rxjs with strict mode. IDE Setup and Manual Scripts ■ VS Code Configuration ■ { "eslint.validate" : ["typescript" , "javascript"] , "editor.codeActionsOnSave" : { "source.fixAll.eslint" : true } } Manual Scripts ■ Script Command Description Check for violations npm run play:state:check Runs the state linter across the project. Generate a report npm run play:state:report Creates a report on unused keys and re-render optimization. Troubleshooting Common Issues ■ Problem: Unused state keys are accumulating in the store. ■ Symptoms : Large store files, properties that are never used. Fix : Run npm run play:state:report and work with your team to prune unused keys. Symptoms : The state seems to change in devtools, but the UI is stale. Fix : This is almost always a direct state mutation. Ensure you are using the spread syntax ({...state, ...newState}) or the set function provided by the store, which uses immer to handle immutability for you. Problem: Performance issues with large state. ■ Symptoms : Slow re-renders, high memory usage. Fix : Use selectors to subscribe to only the necessary state, and consider splitting large stores into smaller, focused stores. Problem: State is not persisting across page reloads. ■ Symptoms : State resets when the page is refreshed. Fix : Implement state persistence using the persistence middleware or localStorage integration. Integration with Other Play+ Systems ■ Logging Integration ■ // State mutations are automatically logged const useAuthStore = createPlayStore < AuthState > ((set) => ({ user : null , setUser : (user) => { playlog . info ("User state updated" , { userId : user ?. id }) ; set ({ user }) ; } , })) ; Error Handling Integration ■ // Handle state errors gracefully const setUser = (user : User) => { try { useAuthStore . getState ( ) . setUser (user) ; } catch (error) { playerror . report (error , { component : "AuthStore" , action : "setUser" , }) ; } } ; Performance Integration ■ // Monitor state performance const useOptimizedStore = createPlayStore < State > ((set) => ({ // ... state }) , { middleware : [(store) => (next) => (action) => { const start = performance . now ( ) ; const result = next (action) ; const duration = performance . now ( ) - start ; if (duration > 5) { playperf . warn ("Slow state action" , { action , duration }) ; } return result ; } ,] , }) ; Migration Guide ■ From Manual Zustand Store ■ // Before import { create } from "zustand" ; const useStore = create ((set) => ({ count : 0 , increment : ( ) => set ((state) => ({ count : state . count + 1 })) , })) ; // After import { createPlayStore } from "@playplus/state" ; const useStore = createPlayStore ((set) => ({ count : 0 , increment : ( ) => set ((state) => ({ count : state . count + 1 })) , })) ; From Redux ■ // Before (Redux) const initialState = { count : 0 } ; const counterReducer = (state = initialState , action) => { switch (action . type) { case "INCREMENT" : return { ... state , count : state . count + 1 } ; default : return state ; } } ; // After (Play+ State) const useCounterStore = createPlayStore ((set) => ({ count : 0 , increment : ( ) => set ((state) => ({ count : state . count + 1 })) , })) ; From

Angular Services ■ // Before @ Injectable ( ) export class StateService { private state = new BehaviorSubject ({ count : 0 }) ; state$ = this . state . asObservable ( ) ; increment ( ) { this . state . next ({ count : this . state . value . count + 1 }) ; } } // After @ Injectable ( ) export class StateService extends PlayStateService < { count : number } > { constructor ( ) { super ({ count : 0 }) ; } increment ( ) { this . dispatch ("increment") ; } } Async/Server State ■ While this guide focuses on client state, Play+ recommends using a dedicated library like TanStack Query (React Query) for managing server cache, which is a different type of state. Our helpers are fully compatible with this approach. Integration with TanStack Query ■ // Use Play+ state for client state const useAuthStore = createPlayStore < AuthState > ((set) => ({ user : null , setUser : (user) => set ({ user }) , })) ; // Use TanStack Query for server state const useUsers = ( ) => { return useQuery ({ queryKey : ["users"] , queryFn : ( ) => fetchUsers ( ) , }) ; } ; Developer Checklist ■ Is my global state defined in the stores (React) or core/services (Angular) directory? Am I avoiding storing derived data in my state? (e.g., calculating fullName from firstName and lastName in the component instead of storing it). In React components, am I using selectors to subscribe to the smallest piece of state necessary? Are all state mutations happening through dedicated actions/methods, not by direct manipulation? Have I considered if this piece of state truly needs to be global, or can it be local component state? Am I using TypeScript interfaces for all state shapes? Have I tested my state actions and selectors? Am I monitoring state performance and re-renders? Have I implemented proper error handling for state operations? Am I using the persistence middleware if state needs to survive page reloads? Summary ■ The Play+ state management system provides: Predictable State : Immutable state with clear update patterns Performance Optimized : Efficient selectors and minimal re-renders Developer Friendly : Simple API that enforces best practices Type Safe : Full TypeScript support with strict typing Debugging Ready : Built-in logging and devtools integration Testing Ready : Easy to test actions and selectors Framework Agnostic : Works with React, Angular, and other frameworks State management should be boring. Focus on your business logic, not the plumbing.